Fast Vertical Mining of Sequential Patterns Using Co-occurrence Information

Philippe Fournier-Viger[1], Antonio Gomariz[2], Manuel Campos[2], and Rincy Thomas[3]

[1] Dept. of Computer Science, University of Moncton, Canada
[2] Dept. of Information and Communication Engineering, University of Murcia, Spain
[3] Dept. of Computer Science, SCT, Bhopal, India
philippe.fournier-viger@umoncton.ca, {agomariz,manuelcampos}@um.es, rinc_thomas@rediffmail.com

Abstract. Sequential pattern mining algorithms using a vertical representation are the most efficient for mining sequential patterns in dense or long sequences, and have excellent overall performance. The vertical representation allows generating patterns and calculating their supports without performing costly database scans. However, a crucial performance bottleneck of vertical algorithms is that they use a generate-candidate-and-test approach that can generate a large amount of infrequent candidates.To address this issue, we propose pruning candidates based on the study of item co-occurrences. We present a new structure named CMAP (Co-occurence MAP) for storing co-occurrence information. We explain how CMAP can be used to prune candidates in three state-of-the-art vertical algorithms, namely SPADE, SPAM and ClaSP. An extensive experimental study with six real-life datasets shows that (1) co-occurrence-based pruning is effective, (2) CMAP is very compact and that (3) the resulting algorithms outperform state-of-the-art algorithms for mining sequential patterns (GSP, PrefixSpan, SPADE and SPAM) and closed sequential patterns (ClaSP and CloSpan).

Keywords: sequential pattern mining, vertical database format, candidate pruning.

1 Introduction

Mining useful patterns in sequential data is a challenging task. Many studies have been proposed for mining interesting patterns in sequence databases [9]. Sequential pattern mining is probably the most popular research topic among them. A subsequence is called *sequential pattern* or *frequent sequence* if it frequently appears in a sequence database, and its frequency is no less than a user-specified minimum support threshold *minsup* [1]. Sequential pattern mining plays an important role in data mining and is essential to a wide range of applications such as the analysis of web click-streams, program executions, medical data, biological data and e-learning data [9]. Several efficient algorithms have been proposed

V.S. Tseng et al. (Eds.): PAKDD 2014, Part I, LNAI 8443, pp. 40–52, 2014.

for sequential pattern mining such as ClaSP [7], CloSpan [12], GSP [11], PrefixSpan [10], SPADE [13] and SPAM [3]. Sequential pattern mining algorithms can be categorized as using a *horizontal database format* (e.g. CloSpan, GSP and PrefixSpan) or a *vertical database format* (e.g. ClaSP, SPADE, SPAM). Using the vertical format provides the advantage of generating patterns and calculating their supports without performing costly database scans [3,7,13]. This allows vertical algorithms to perform better on datasets having dense or long sequences than algorithms using the horizontal format, and to have excellent overall performance [2,3,7]. However, a crucial performance bottleneck of vertical algorithms is that they use a generate-candidate-and-test approach, which can generate a large amount of patterns that do not appear in the input database or are infrequent. An important research questions that arises is: Could we design an effective candidate pruning method for vertical mining algorithms to improve mining performance? Answering this question is challenging. It requires designing a candidate pruning mechanism (1) that is effective at pruning candidates and (2) that has a small runtime and memory cost. Moreover, the mechanism should preferably be generic. i.e. applicable to any vertical mining algorithms.

In this paper, we present a solution to this issue based on the study of item co-occurrences. Our contribution is threefold. First, to store item co-occurrence information, we introduce a new data structure named *Co-occurrence MAP* (CMAP). CMAP is a small and compact structure, which can be built with a single database scan.

Second, we propose a generic candidate pruning mechanism for vertical sequential pattern mining algorithms based on the CMAP data structure. We describe how the pruning mechanism is integrated in three state-of-the-art algorithms ClaSP, SPADE and SPAM. We name the resulting algorithms CM-ClaSP, CM-SPADE and CM-SPAM.

Third, we perform a wide experimental evaluation on six real-life datasets. We compare the performance of CM-ClaSP, CM-SPADE and CM-SPAM with state-of-the-art algorithms for mining sequential patterns (GSP, PrefixSpan, SPADE and SPAM) and closed sequential patterns (ClaSP and CloSpan). Results show that the modified algorithms (1) prune a large amount of candidates, (2) and are up to eight times faster than the corresponding original algorithms and (3) that CM-ClaSP and CM-SPADE have respectively the best performance for sequential pattern mining and closed sequential pattern mining.

The rest of the paper is organized as follows. Section 2 defines the problem of sequential pattern mining and reviews the main characteristics of ClaSP, SPADE and SPAM. Section 3 describes the CMAP structure, the pruning mechanism, and how it is integrated in ClaSP, SPADE and SPAM. Section 4 presents the experimental study. Finally, Section 5 presents the conclusion.

2 Problem Definition and Related Work

Definition 1 (sequence database). Let $I = \{i_1, i_2, ..., i_l\}$ be a set of items (symbols). An *itemset* $I_x = \{i_1, i_2, ..., i_m\} \subseteq I$ is an unordered set of distinct

items. The *lexicographical order* $\succ_{lex}$ is defined as any total order on I. Without loss of generality, it is assumed in the following that all itemsets are ordered according to $\succ_{lex}$. A *sequence* is an ordered list of itemsets $s = \langle I_1, I_2, ..., I_n \rangle$ such that $I_k \subseteq I$ $(1 \leq k \leq n)$. A *sequence database* SDB is a list of sequences $SDB = \langle s_1, s_2, ..., s_p \rangle$ having sequence identifiers (SIDs) $1, 2...p$. **Example.** A sequence database is shown in Fig. 1 (left). It contains four sequences having the SIDs 1, 2, 3 and 4. Each single letter represents an item. Items between curly brackets represent an itemset. The first sequence $\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$ contains five itemsets. It indicates that items a and b occurred at the same time, were followed by c, then f, g and lastly e.

SID	Sequences
1	⟨{*a, b*},{*c*},{*f, g*},{*g*},{*e*}⟩
2	⟨{*a, d*},{*c*},{*b*},{*a, b, e, f*}⟩
3	⟨{*a*},{*b*},{*f*},{*e*}⟩
4	⟨{*b*},{*f, g*}⟩

ID	Pattern	Support
p1	⟨{*a*},{*f*}⟩	3
p2	⟨{*a*},{c} {*f*}⟩	2
p3	⟨{*b*},{f,g}⟩	2
p4	⟨{*g*},{e}⟩	2
p5	⟨{*c*},{f}⟩	2
p6…	⟨{*b*}⟩	4

Fig. 1. A sequence database (left) and some sequential patterns found (right)

Definition 2 (sequence containment). A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is said to be *contained in* a sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$ iff there exist integers $1 \leq i_1 < i_2 < ... < i_n \leq m$ such that $A_1 \subseteq B_{i1}, A_2 \subseteq B_{i2}, ..., A_n \subseteq B_{in}$ (denoted as $s_a \sqsubseteq s_b$). **Example.** Sequence 4 in Fig. 1 (left) is contained in Sequence 1.

Definition 3 (prefix). A sequence $s_a = \langle A_1, A_2, ..., A_n \rangle$ is a *prefix* of a sequence $s_b = \langle B_1, B_2, ..., B_m \rangle$, $\forall n < m$, iff $A_1 = B_1, A_2 = B_2, ..., A_{n-1} = B_{n-1}$ and the first $|A_n|$ items of B_n according to $\succ_{lex}$ are equal to A_n.

Definition 4 (support). The *support* of a sequence s_a in a sequence database SDB is defined as the number of sequences $s \in SDB$ such that $s_a \sqsubseteq s$ and is denoted by $sup_{SDB}(s_a)$.

Definition 5 (sequential pattern mining). Let *minsup* be a threshold set by the user and SDB be a sequence database. A sequence s is a *sequential pattern* and is deemed *frequent* iff $sup_{SDB}(s) \geq minsup$. The *problem of mining sequential patterns* is to discover all sequential patterns [11]. **Example.** Fig. 1 (right) shows 6 of the 29 sequential patterns found in the database of Fig. 1 (left) for $minsup = 2$.

Definition 6 (closed sequential pattern mining). A sequential pattern s_a is said to be *closed* if there is no other sequential pattern s_b, such that s_b is a superpattern of s_a, $s_a \sqsubseteq s_b$, and their supports are equal. The problem of *closed sequential patterns* is to discover the set of closed sequential patterns, which is a compact summarization of all sequential patterns [7,12].

Definition 7 (horizontal database format). A *sequence database in horizontal format* is a database where each entry is a sequence. **Example.** Fig. 1 (left) shows an horizontal sequence database.

Definition 8 (vertical database format). A *sequence database in vertical format* is a database where each entry represents an item and indicates the list of sequences where the item appears and the position(s) where it appears [13]. **Example.** Fig. 2 shows the vertical representation of the database of Fig. 1 (left).

From the vertical representation, a structure named *IdList* [13] can be associated with each pattern. IdLists allow calculating the support of a pattern quickly by making join operations with IdLists of smaller patterns. To discover sequential patterns using IdLists, a single database scan is required to create IdLists of patterns containing a single items, since IdList of larger patterns are obtained by performing the aforementioned join operation (see [13] for details). Several works proposed alternative representations for IdLists to save time in join operations, being the bitset representation the most efficient one [3,2].

a	
SID	Itemsets
1	1
2	1,4
3	1
4	

b	
SID	Itemsets
1	1
2	3,4
3	2
4	1

c	
SID	Itemsets
1	2
2	2
3	
4	

d	
SID	Itemsets
1	
2	1
3	
4	

e	
SID	Itemsets
1	5
2	4
3	4
4	

f	
SID	Itemsets
1	3
2	4
3	3
4	2

g	
SID	Itemsets
1	3,4
2	
3	
4	2

Fig. 2. The vertical representation of the example database shown in Figure 1(left)

The horizontal format is used by Apriori-based algorithms (e.g. GSP) and pattern-growth algorithms (e.g. CloSpan and PrefixSpan). The two main algorithms using the vertical database format are SPADE and SPAM. Other algorithms are variations such as bitSPADE [2] and ClaSP [7]. SPADE and SPAM differ mainly by their candidate generation process, which we review thereafter.

Candidate Generation in SPAM. The pseudocode of SPAM is shown in Fig. 3. SPAM take as input a sequence database SDB and the *minsup* threshold. SPAM first scans the input database SDB once to construct the vertical representation of the database $V(SDB)$ and the set of frequent items F_1. For each frequent item $s \in F_1$, SPAM calls the SEARCH procedure with $\langle s \rangle$, F_1, $\{e \in F_1 | e \succ_{lex} s\}$, and *minsup*. The SEARCH procedure outputs the pattern $\langle \{s\} \rangle$ and recursively explore candidate patterns starting with the prefix $\langle \{s\} \rangle$. The SEARCH procedure takes as parameters a sequential pattern *pat* and two

sets of items to be appended to *pat* to generate candidates. The first set S_n represents items to be appended to *pat* by *s*-extension. The *s*-extension of a sequential pattern $\langle I_1, I_2, ...I_h\rangle$ with an item x is defined as $\langle I_1, I_2, ...I_h, \{x\}\rangle$. The second set S_i represents items to be appended to *pat* by *i*-extension. The *i*-extension of a sequential pattern $\langle I_1, I_2, ...I_h\rangle$ with an item x is defined as $\langle I_1, I_2, ...I_h \cup \{x\}\rangle$. For each candidate *pat* generated by an extension, SPAM calculate its support to determine if it is frequent. This is done by making a join operation (see [3] for details) and counting the number of sequences where the pattern appears. The IdList representation used by SPAM is based on bitmaps to get faster operations [3]. If the pattern *pat* is frequent, it is then used in a recursive call to SEARCH to generate patterns starting with the prefix *pat*. Note that in the recursive call, only items that resulted in a frequent pattern by extension of *pat* are considered for extending *pat*. SPAM prunes the search space by not extending infrequent patterns. This can be done due to the property that an infrequent sequential pattern cannot be extended to form a frequent pattern [1].

SPAM(*SDB, minsup*)
1. Scan *SDB* to create *V*(*SDB*) and identify F_1, the list of frequent items.
2. **FOR** each item s $\in F_1$,
3. **SEARCH**($\langle s\rangle$, F_1, $\{e \in F_1 \mid e \succ_{lex} s\}$, *minsup*).

SEARCH(*pat*, S_n, I_n, *minsup*)
1. Output pattern *pat*.
2. $S_{temp} := I_{temp} := \emptyset$
3. **FOR** each item $j \in S_n$,
4. **IF** the s-extension of *pat* is frequent **THEN** $S_{temp} := S_{temp} \cup \{i\}$.
5. **FOR** each item $j \in S_{temp}$,
6. **SEARCH**(the s-extension of *pat* with *j*, S_{temp}, $\{e \in S_{temp} \mid e \succ_{lex} j\}$, *minsup*).
7. **FOR** each item $j \in I_n$,
8. **IF** the i-extension of *pat* is frequent **THEN** $I_{temp} := I_{temp} \cup \{i\}$.
9. **FOR** each item $j \in I_{temp}$,
10. **SEARCH**(i-extension of *pat* with *j*, S_{temp}, $\{e \in I_{temp} \mid e \succ_{lex} j\}$, *minsup*).

Fig. 3. The pseudocode of SPAM

Candidate Generation in SPADE. The pseudocode of SPADE is shown in Fig. 4. The SPADE procedure takes as input a sequence database SDB and the $minsup$ threshold. SPADE first constructs the vertical database $V(SDB)$ and identifies the set of frequent sequential patterns F_1 containing frequent items. Then, SPADE calls the ENUMERATE procedure with the equivalence class of size 0. An *equivalence class* of size k is defined as the set of all frequent patterns containing k items sharing the same prefix of $k-1$ items. There is only an equivalence class of size 0 and it is composed of F_1. The ENUMERATE procedure receives an equivalence class F as parameter. Each member A_i of the equivalence class is output as a frequent sequential pattern. Then, a set T_i, representing the equivalence class of all frequent extensions of A_i is initialized to the empty set. Then, for each pattern $A_j \in F$ such that $i \succ_{lex} j$, the pattern A_i is merged with A_j to form larger pattern(s). For each such pattern r, the support

of r is calculated by performing a join operation between IdLists of A_i and A_j. If the cardinality of the resulting IdList is no less than $minsup$, it means that r is a frequent sequential pattern. It is thus added to T_i. Finally, after all pattern A_j have been compared with A_i, the set T_i contains the whole equivalence class of patterns starting with the prefix A_i. The procedure ENUMERATE is then called with T_i to discover larger sequential patterns having A_i as prefix. When all loops terminate, all frequent sequential patterns have been output (see [13] for the proof that this procedure is correct and complete).

SPADE and SPAM are very efficient for datasets having dense or long sequences and have excellent overall performance since performing join operations to calculate the support of candidates does not require scanning the original database unlike algorithms using the horizontal format. For example, the well-known PrefixSpan algorithm, which uses the horizontal format, performs a database projection for each item of each frequent sequential pattern, in the worst case, which is extremely costly. The main performance bottleneck of vertical mining algorithms is that they use a generate-candidate-and-test approach and therefore spend lot of time evaluating patterns that do not appear in the input database or are infrequent. In the next section, we present a novel method based on the study of item co-occurrence information to prune candidates generated by vertical mining algorithms to increase their performance.

```
SPADE(SDB, minsup)
1.  Scan SDB to create V(SDB) and identify F1 the list of frequent items.
2.  ENUMERATE(F1).
```

```
ENUMERATE(an equivalence class F)
1.  FOR each pattern Ai ∈ F
2.      Output Ai.
3.      Ti := ∅.
4.      FOR each pattern Aj ∈ F, with j ≥ i
5.          R = MergePatterns(Ai, Aj)
6.          FOR each pattern r ∈ R
7.              IF sup(R) ≥ minsup THEN
8.                  Ti := Ti ∪{R};
9.      ENUMERATE(Ti)
```

Fig. 4. The pseudocode of SPADE

3 Co-occurrence Pruning

In this section, we introduce our approach, consisting of a data structure for storing co-occurrence information, and its properties for candidate pruning for vertical sequential pattern mining. Then, we describe how the data structure is integrated in three state-of-the-art vertical mining algorithms, namely ClaSP, SPADE and SPAM.

3.1 The Co-occurrence Map

Definition 9. An item k is said to *succeed by i-extension* to an item j in a sequence $\langle I_1, I_2, ..., I_n \rangle$ iff $j, k \in I_x$ for an integer x such that $1 \leq x \leq n$ and $k \succ_{lex} j$.

Definition 10. An item k is said to *succeed by s-extension* to an item j in a sequence $\langle I_1, I_2, ..., I_n \rangle$ iff $j \in I_v$ and $k \in I_w$ for some integers v and w such that $1 \leq v < w \leq n$.

Definition 11. A *Co-occurrence MAP* (CMAP) is a structure mapping each item $k \in I$ to a set of items succeeding it. We define two CMAPs named $CMAP_i$ and $CMAP_s$. $CMAP_i$ maps each item k to the set $cm_i(k)$ of all items $j \in I$ succeeding k by i-extension in no less than $minsup$ sequences of SDB. $CMAP_s$ maps each item k to the set $cm_s(k)$ of all items $j \in I$ succeedings k by s-extension in no less than $minsup$ sequences of SDB. **Example.** The CMAP structures built for the sequence database of Fig. 1(left) are shown in Table 1, being $CMAP_i$ on the left part and $CMAP_s$ on the right part. Both tables have been created considering a $minsup$ of two sequences. For instance, for the item f, we can see that it is associated with an item, $cm_i(f) = \{g\}$, in $CMAP_i$, whereas it is associated with two items, $cm_s(f) = \{e, g\}$, in $CMAP_s$. This indicates that both items e and g succeed to f by s-extension and only item g does the same for i-extension, being all of them in no less than $minsup$ sequences.

Table 1. $CMAP_i$ and $CMAP_s$ for the database of Fig. 1 and $minsup = 2$

$CMAP_i$	
item	is succeeded by (i-extension)
a	$\{b\}$
b	$\emptyset$
c	$\emptyset$
e	$\emptyset$
f	$\{g\}$
g	$\emptyset$

$CMAP_s$	
item	is succeeded by (s-extension)
a	$\{b, c, e, f\}$
b	$\{e, f, g\}$
c	$\{e, f\}$
e	$\emptyset$
f	$\{e, g\}$
g	$\emptyset$

Size Optimization. Let $n = |I|$ be the number of items in SDB. To implement a CMAP, a simple solution is to use an $n \times n$ matrix (two-dimensional array) M where each row (column) correspond to a distinct item and such that each entry $m_{j,k} \in M$ represents the number of sequences where the item k succeed to the item i by i-extension or s-extension. The size of a CMAP would then be $O(n^2)$. However, the size of CMAP can be reduced using the following strategy. It can be observed that each item is succeeded by only a small subset of all items for most datasets. Thus, few items succeed another one by extension, and thus, a CMAP may potentially waste large amount of memory for empty entries if we consider them by means of a $n \times n$ matrix. For this reason, in our implementations we instead implemented each CMAP as a hash table of hash sets, where an hashset corresponding to an item k only contains the items that succeed to k in at least $minsup$ sequences.

3.2 Co-occurrence-Based Pruning

The CMAP structure can be used for pruning candidates generated by vertical sequential pattern mining algorithms based on the following properties.

Property 1 (pruning an i-extension). Let be a frequent sequential pattern A and an item k. If there exists an item j in the last itemset of A such that k *does not* belongs to $cm_i(j)$, then the i-extension of A with k is infrequent. **Proof.** If an item k does not appear in $cm_i(j)$, then k succeed to j by i-extension in less than $minsup$ sequences in the database SDB. It is thus clear that appending k by i-extension to a pattern A containing j in its last itemset will not result in a frequent pattern. □

Property 2 (pruning an s-extension). Let be a frequent sequential pattern A and an item k. If there exists an item $j \in A$ such that the item k *does not* belongs to $cm_s(j)$, then the s-extension of A with k is infrequent. **Proof.** If an item k does not appear in $cm_s(j)$, then k succeeds to j by s-extension in less than $minsup$ sequences from the sequence database SDB. It is thus clear that appending j by s-extension to a pattern A containing k will not result in a frequent pattern. □

Property 3 (pruning a prefix). The previous properties can be generalized to prune all patterns starting with a given prefix. Let be a frequent sequential pattern A and an item k. If there exists an item $j \in A$ (equivalently j in the last itemset of A) such that there is an item $k \in cm_s(j)$ (equivalently in $cm_i(j)$), then all supersequences B having A as prefix and where k succeeds j by s-extension (equivalently i-extension to the last itemset) in A in B are infrequent. **Proof.** If an item k does not appear in $cm_s(j)$ (equivalently $cm_i(j)$), therefore k succeeds to j in less than $minsup$ sequences by s-extension (equivalently i-extension to the last itemset) in the database SDB. It is thus clear that no frequent pattern containing j (equivalently j in the last itemset) can be formed such that k is appended by s-extension (equivalently by i-extension to the last itemset). □

3.3 Integrating Co-occurrence Pruning in Vertical Mining

Integration in SPADE. The integration in SPADE is done as follows. In the ENUMERATE procedure, consider a pattern r obtained by merging two patterns $A_i = P \cup x$ and $A_j = P \cup y$, being P a common prefix for A_i and A_j. Let y be the item that is appended to A_i to generate r. If r is an i-extension, we use the $CMAP_i$ structure, otherwise, if r is an s-extension, we use $CMAP_s$. If the last item a of r does not have an item $x \in cm_i(a)$ (equivalently in $cm_s(a)$), then the pattern r is infrequent and r can be immediately discarded, avoiding the join operation to calculate the support of r. This pruning strategy is correct based on Properties 1, 2 and 3.

Note that to perform the pruning in SPADE, we do not have to check if items of the prefix P are succeeded by the item $y \in A_j$. This is because the items of P are also in A_j. Therefore, checking the extension of P by y was already done, and it is not necessary to do it again.

Integration in SPAM. The CMAP structures are used in the SEARCH procedure as follows. Let a sequential pattern *pat* being considered for *s*-extension ($x \in S_n$) or *i*-extension ($x \in S_i$) with an item x (line 3). If the last item a in *pat* does not have an item $x \in cm_s(a)$ (equivalently cm_i), then the pattern resulting from the extension of *pat* with x will be infrequent and thus the join operation of x with *pat* to count the support of the resulting pattern does not need to be performed (by Property 1 and 2). Furthermore, the item x should not be considered for generating any pattern by *s*-extension (*i*-extension) having *pat* as prefix (by Property 3). Therefore x should not be added to the variable S_{temp} (I_{temp}) that is passed to the recursive call to the SEARCH procedure.

Note that to perform the pruning in SPAM, we do not have to check for extensions of *pat* with x for all the items since such items, except for the last one, have already been checked for extension in previous steps.

Integration in ClaSP. We have also integrated co-occurrence pruning in ClaSP [7], a state of the art algorithm for closed sequential pattern mining. The integration in ClaSP is not described here since it is done as in SPAM since ClaSP is based on SPAM.

4 Experimental Evaluation

We performed experiments to assess the performance of the proposed algorithms. Experiments were performed on a computer with a third generation Core i5 processor running Windows 7 and 5 GB of free RAM. We compared the performance of the modified algorithms (CM-ClaSP, CM-SPADE, CM-SPAM) with state-of-the-art algorithms for sequential pattern mining (GSP, PrefixSpan, SPADE and SPAM) and closed sequential pattern mining (ClaSP and CloSpan). All algorithms were implemented in Java. Note that for SPADE algorithms, we use the version proposed in [2] that implement IdLists by means of bitmaps. All memory measurements were done using the Java API. Experiments were carried on six real-life datasets having varied characteristics and representing four different types of data (web click stream, text from books, sign language utterances and protein sequences). Those datasets are *Leviathan*, *Sign*, *Snake*, *FIFA*, *BMS* and *Kosarak10k*. Table 2 summarizes their characteristics. The source code of all algorithms and datasets used in our experiments can be downloaded from `http://goo.gl/hDtdt`.

The experiments consisted of running all the algorithms on each dataset while decreasing the *minsup* threshold until algorithms became too long to execute, ran out of memory or a clear winner was observed. For each dataset, we recorded the execution time, the percentage of candidate pruned by the proposed algorithms and the total size of CMAPs. The comparison of execution times is shown in Fig. 5. The percentage of candidates pruned by the proposed algorithms is shown in Table 3.

Effectiveness of Candidate Pruning. CM-ClaSP, CM-SPADE and CM-SPAM are generally from about 2 to 8 times faster than the corresponding original algorithms (ClaSP, SPADE and SPAM). This shows that co-occurrence

Table 2. Dataset characteristics

dataset	sequence count	distinct item count	avg. seq. length (items)	type of data
Leviathan	5834	9025	33.81 (std= 18.6)	book
Sign	730	267	51.99 (std = 12.3)	language utterances
Snake	163	20	60 (std = 0.59)	protein sequences
FIFA	20450	2990	34.74 (std = 24.08)	web click stream
BMS	59601	497	2.51 (std = 4.85)	web click stream
Kosarak10k	10000	10094	8.14 (std = 22)	web click stream

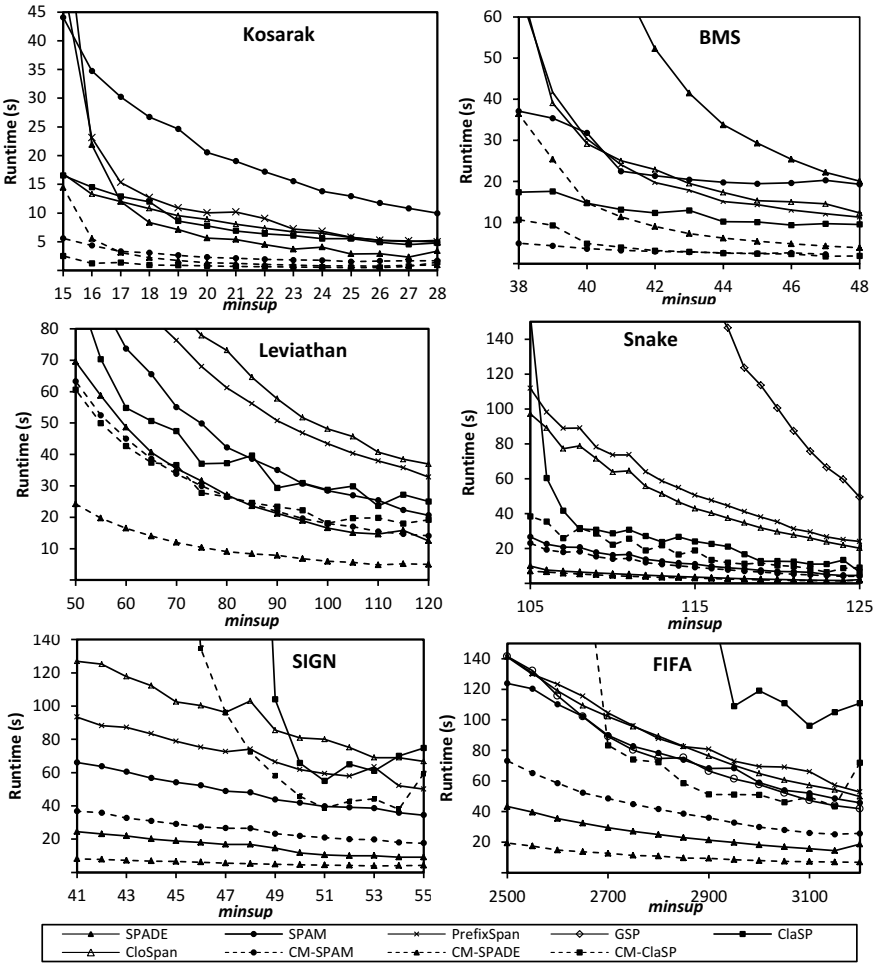

Fig. 5. Execution times

Table 3. Candidate reduction

	BMS	Kosarak	Leviathan	Snake	Sign	Fifa
CM-SPAM	78 to 93 %	94 to 98 %	50 to 51 %	28%	63 %	61 to 68 %
CM-SPADE	75 to 76 %	98 %	50 %	25 to 26 %	69 %	63 to 69 %
CM-ClaSP	79 to 93%	75 %	50 to 52 %	18 %	63 %	67 to 68 %

Table 4. CMAP implementations comparison

	BMS	Kosarak	Leviathan	Snake	Sign	Fifa
minsup	38	16	60	105	43	2500
CMAP Size (hashmap)	**0.5 MB**	**33.1 MB**	**15 MB**	64 KB	3.19 MB	**0.4 MB**
CMAP Size (matrix)	0.9 MB	388 MB	310 MB	**1.7 KB**	**0.2 MB**	34.1 MB
Pair count (hashmap)	50,885	58,772	41,677	144	17,887	2,500
Pair count (matrix)	247,009	101,888,836	81,450,625	400	71,289	8,940,100

pruning is an effective technique to improve the performance of vertical mining algorithms. The dataset where the performance of the modified algorithms is closer to the original algorithms is Snake because all items co-occurs with each item in almost all sequences and therefore fewer candidates could be pruned. For other datasets, the percentage of candidates pruned range from 50% and to 98 %). The percentage slowly decrease as *minsup* get lower because less pairs in CMAP had a count lower than *minsup* for pruning.

Best Performance. For mining sequential patterns, CM-SPADE had the best performance on all but two datasets (Kosarak and BMS). The second best algorithm for mining sequential patterns is CM-SPAM (best performance on BMS and Kosarak). For mining closed sequential patterns, CM-ClaSP has the best performance on four datasets (Kosarak, BMS, Snake and Leviathan). CM-ClaSP is only outperformed by CloSpan on two datasets (FIFA and SIGN) and for low *minsup* values.

Memory Overhead. We also studied the memory overhead of using CMAPs. We measured the total memory used by a matrix implementation and a hashmap implementation of CMAPs (cf. section 3.1) for all datasets for the lowest *minsup* values from the previous experiments. Results are shown in Table 4. Size is measured in terms of memory usage and number of entries in CMAPs. From these results, we conclude that (1) the matrix implementation is smaller for datasets with a small number of distinct items (Snake and SIGN), while (2) the hashmap implementation is smaller for datasets with a large number of items (BMS, Leviathan, Kosarak and FIFA) and (3) the hashmap implementation has a very low memory overhead (less than 35 MB on all datasets).

5 Conclusion

Sequential pattern mining algorithms using the vertical format are very efficient because they can calculate the support of candidate patterns by avoiding costly

database scans. However, the main performance bottleneck of vertical mining algorithms is that they usually spend lot of time evaluating candidates that do not appear in the input database or are infrequent. To address this problem, we presented a novel data structure named CMAP for storing co-occurrence information. We have explained how CMAPs can be used for pruning candidates generated by vertical mining algorithms. We have shown how to integrate CMAPs in three state-of-the-art vertical algorithms. We have performed an extensive experimental study on six real-life datasets to compare the performance of the modified algorithms (CM-ClaSP, CM-SPADE and CM-SAPM) with state-of-the-art algorithms (ClaSP, CloSpan, GSP, PrefixSpan, SPADE and SPAM). Results show that the modified algorithms (1) prune a large amount of candidates, (2) are up to 8 times faster than the corresponding original algorithms and (3) that CM-SPADE and CM-ClaSP have respectively the best performance for mining sequential patterns and closed sequential patterns.

The source code of all algorithms and datasets used in our experiments can be downloaded from `http://goo.gl/hDtdt`.

For future work, we plan to develop additional optimizations and also to integrate them in sequential rule mining [5], top-k sequential pattern mining [4] and maximal sequential pattern mining [6].

Acknowledgement. This work is partially financed by a National Science and Engineering Research Council (NSERC) of Canada research grant, a PhD grant from the Seneca Foundation (Regional Agency for Science and Technology of the Region de Murcia), and by the Spanish Office for Science and Innovation through project TIN2009-14372-C03-01 and PlanE, and the European Union by means of the European Regional Development Fund (ERDF, FEDER).

References

1. Agrawal, R., Ramakrishnan, S.: Mining sequential patterns. In: Proc. 11th Intern. Conf. Data Engineering, pp. 3–14. IEEE (1995)
2. Aseervatham, S., Osmani, A., Viennet, E.: bitSPADE: A Lattice-based Sequential Pattern Mining Algorithm Using Bitmap Representation. In: Proc. 6th Intern. Conf. Data Mining, pp. 792–797. IEEE (2006)
3. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proc. 8th ACM SIGKDD Intern. Conf. Knowledge Discovery and Data Mining, pp. 429–435. ACM (2002)
4. Fournier-Viger, P., Gomariz, A., Gueniche, T., Mwamikazi, E., Thomas, R.: TKS: Efficient Mining of Top-K Sequential Patterns. In: Motoda, H., Wu, Z., Cao, L., Zaiane, O., Yao, M., Wang, W. (eds.) ADMA 2013, Part I. LNCS, vol. 8346, pp. 109–120. Springer, Heidelberg (2013)
5. Fournier-Viger, P., Nkambou, R., Tseng, V.S.: RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth. In: Proc. ACM 26th Symposium on Applied Computing, pp. 954–959 (2011)
6. Fournier-Viger, P., Wu, C.-W., Tseng, V.S.: Mining Maximal Sequential Patterns without Candidate Maintenance. In: Motoda, H., Wu, Z., Cao, L., Zaiane, O., Yao, M., Wang, W. (eds.) ADMA 2013, Part I. LNCS, vol. 8346, pp. 169–180. Springer, Heidelberg (2013)

7. Gomariz, A., Campos, M., Marin, R., Goethals, B.: ClaSP: An Efficient Algorithm for Mining Frequent Closed Sequences. In: Pei, J., Tseng, V.S., Cao, L., Motoda, H., Xu, G. (eds.) PAKDD 2013, Part I. LNCS, vol. 7818, pp. 50–61. Springer, Heidelberg (2013)
8. Han, J., Kamber, M.: Data Mining: Concepts and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2006)
9. Mabroukeh, N.R., Ezeife, C.I.: A taxonomy of sequential pattern mining algorithms. ACM Computing Surveys 43(1), 1–41 (2010)
10. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.: Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans. Knowledge Data Engineering 16(11), 1424–1440 (2004)
11. Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalizations and Performance Improvements. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 3–17. Springer, Heidelberg (1996)
12. Yan, X., Han, J., Afshar, R.: CloSpan: Mining closed sequential patterns in large datasets. In: Proc. 3rd SIAM Intern. Conf. on Data Mining, pp. 166–177 (2003)
13. Zaki, M.J.: SPADE: An efficient algorithm for mining frequent sequences. Machine Learning 42(1), 31–60 (2001)